猫狗分类实验报告

姓名：赵巍山　　学号：202228019427035　　培养单位：空天信息创新研究院

一、 实验目的：

1. 进一步理解和掌握卷积神经网络中卷积层、卷积步长、卷积核、池化层、池 化核、微调(Fine-tune)等概念。
2. 进一步掌握使用深度学习框架进行图像分类任务的具体流程：如读取数据、 构造网络、训练和测试模型等等。

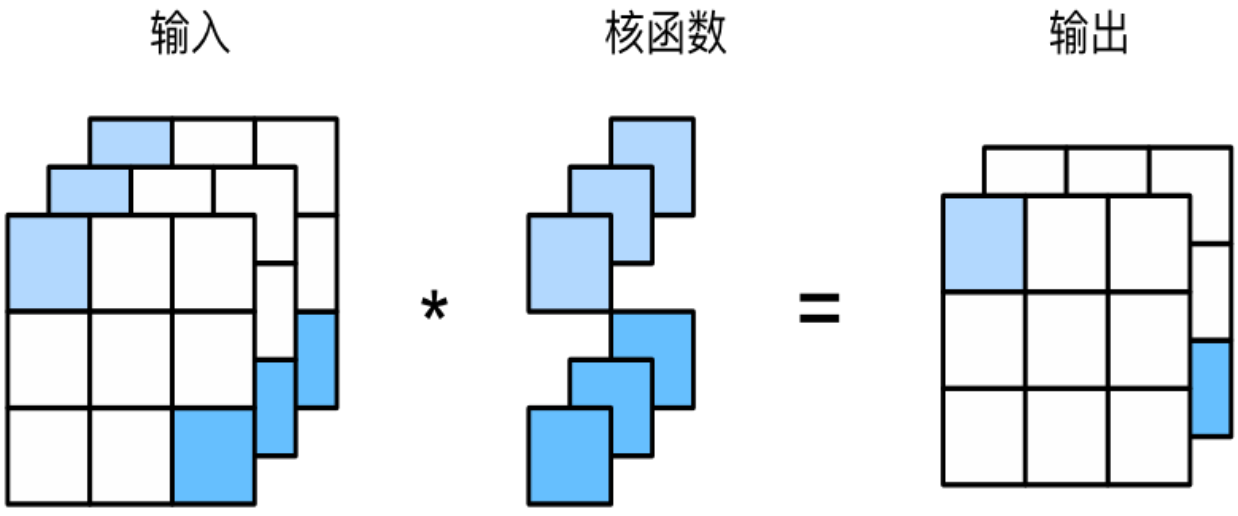
二、 实验要求：

1. 基于 Python 语言和任意一种深度学习框架（实验指导书中使用 Pytorch 框架进行介绍），从零开始一步步完成数据读取、网络构建、模型训练和模型 测试等过程，最终实现一个可以进行猫狗图像分类的分类器。
2. 使用 Kaggle 猫狗竞赛的 原始数据集，原则上要求人为划分的数据集中，训练集图像总数不少于 2000 张，测试集图像总数不少于大于 500，最终模型的 准确率要求不低于 75%。

三、 实验原理：

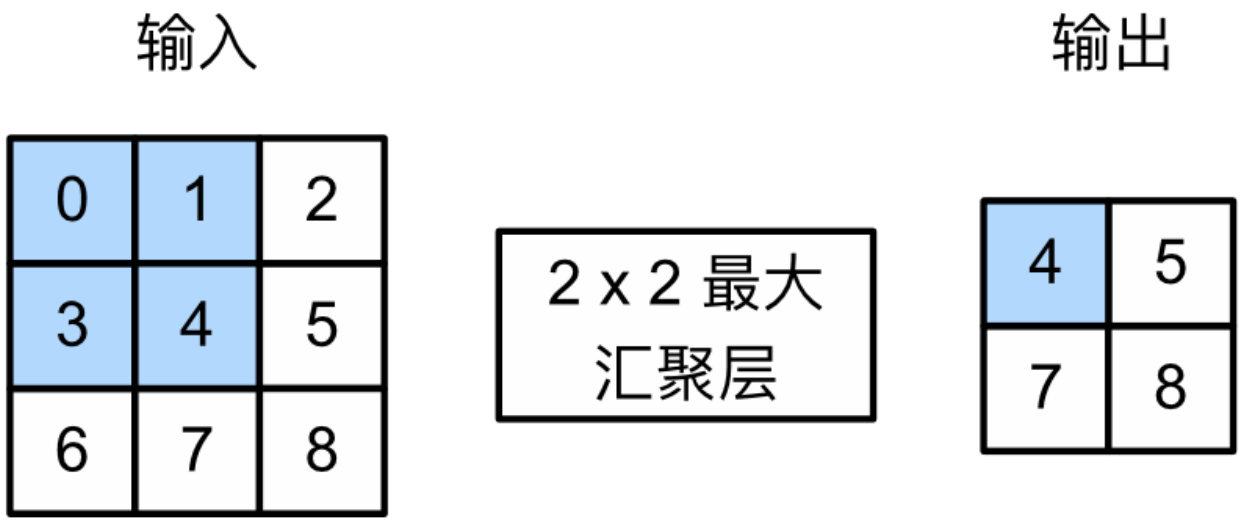
卷积层：

卷积层通过滑动一个固定大小的窗口，称为卷积核或过滤器，对输入数据进行滤波操作，提取出数据中的局部特征。卷积操作可以捕捉输入数据的局部相关性，减少需要训练的参数数量，增强了神经网络对平移、旋转、缩放等操作的鲁棒性。



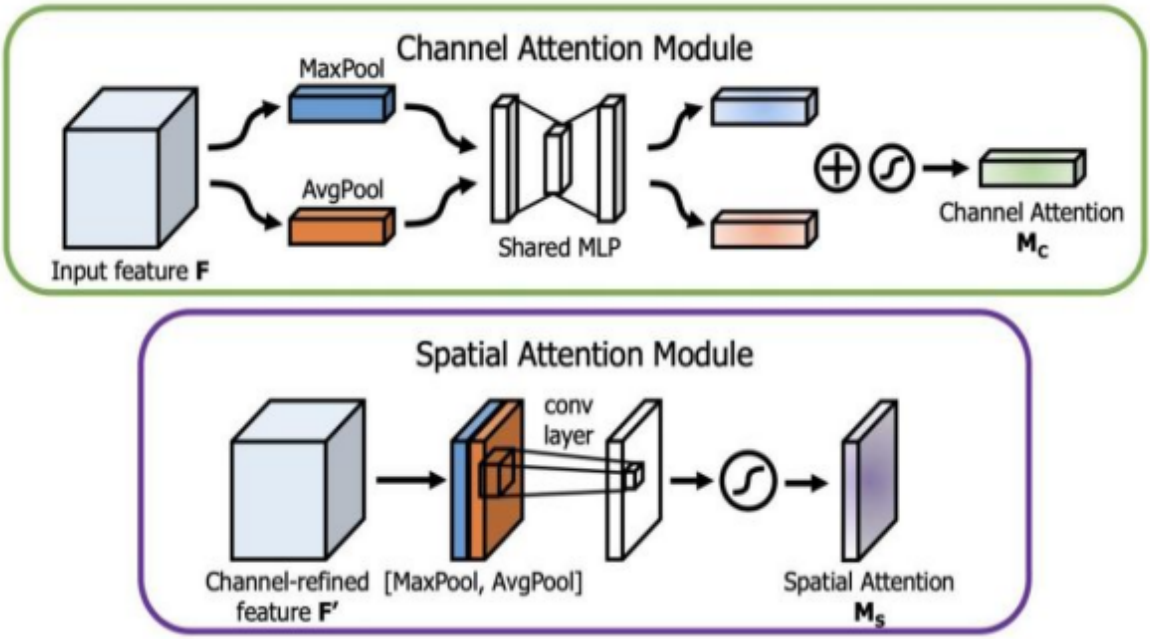
池化层：

用于降低特征图的空间分辨率，减少需要训练的参数数量，提高网络的鲁棒性和泛化能力。池化操作通常在卷积层之后进行，它可以对卷积层的输出进行空间上的降采样，从而减少输出特征图的大小。



注意力机制CBAM：

注意力机制（Attention Mechanism）作为机器学习的一种数据的处理方法，对于我们提取图像目标特征具有十分重要的作用。其本质在于其可以图像的特定部分进行集中观测，进而获取出该特征的诸多信息。在实现的方式上来看，其主要有俩部分构成：注意力打分部分以及注意力聚焦的部分，主要是通过对输入特征矩阵进行多维的打分和加权，评价当前所需要关注点的评分,并将此评分以加权的形式叠加到检测网络中，从而实现了突出图像信息中的关键特征的作用，使得目标的特征获取能力更强，特征区域关注度更高，使得网络结构更好的检测提取目标的特征信息。



处理流程：

通过卷积层进行特征信息的提取，并将提取的信息通过池化层进行下采样操作，减少参数数量，此外通过全连接层来处理输出的维度，使其最终与我们的目的的输出相匹配。

评价指标：

基于样本预测值和真实值是否相符，可得到4种结果：

```
TP(True Positive):样本预测值与真实值相符且均为正，即真阳性
FP(False Positive):样本预测值为正而真实值为负，即假阳性
FN(False Negative):样本预测值为负而真实值为正，即假阴性
TN(True Negative):样本预测值与真实值相符且均为负，即真阴性
```

准确率：

$$ACC = \frac{TP + TN}{TP + TN + FN + FP}$$

四、 实验原理步骤：

1.导入关键的库

```
import os
import sys
import json

import torch
import torch.nn as nn
from torchvision import transforms, datasets, utils
import matplotlib.pyplot as plt
import numpy as np
import torch.optim as optim
from tqdm import tqdm
import torch.optim as optim
from mymodel import MYnet
```

2.指定运算的设备

```
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")#判断设备是否GPU可以用，否则就指定设备为CPU
    print("using {} device.".format(device))
```

3.数据的加载和预处理

```
#数据的预处理，其中包括了数据的随机裁剪，标准化处理等，将数据转换成tensor的格式。
data_transform = {

        "train": transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(), #将数据利用transform方法进行预处理
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),
        "val": transforms.Compose([transforms.Resize((224, 224)),  # cannot 224, must (224, 224)
```

```
                                    transforms.ToTensor(), #将数据利用transform方法进行
预处理
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5))])}
#从文件夹中读取数据集的图片，其下面的文件夹的名字为类别。
    train_dataset= datasets.ImageFolder('dataset\\train',data_transform["train"])
    train_num = len(train_dataset)
    flower_list = train_dataset.class_to_idx
    cla_dict = dict((val, key) for key, val in flower_list.items())
    # write dict into json file
    json_str = json.dumps(cla_dict, indent=4)
    with open('class_indices.json', 'w') as json_file:
        json_file.write(json_str)
#指定训练的batch_size为16
    batch_size =16
    nw = 0
    print('Using {} dataloader workers every process'.format(nw))
#将数据通过dataloader加载成按照batch划分的数据
    train_loader = torch.utils.data.DataLoader(train_dataset,
                                               batch_size=batch_size, shuffle=True,
                                               num_workers=nw)

    validate_dataset = datasets.ImageFolder('dataset\\val', data_transform["val"])
    val_num = len(validate_dataset)
    train_num = len(train_dataset)
    validate_loader = torch.utils.data.DataLoader(validate_dataset,
                                                  batch_size=4, shuffle=True,
                                                  num_workers=nw)
    print("using {} images for training, {} images for
validation.".format(train_num, val_num))
```

4.模型的定义

```
import torch.nn as nn
import torch.nn.functional as F
import torch
#模型的定义，定义一个关键的类
class MYnet(nn.Module):
       def __init__(self,clss_num=2,init_weights=False):
           super(MYnet,self).__init__()
          #定义第一个特征提取的模块，此处采用nn.sequential来叠加我们的层
           self.features=nn.Sequential(
               nn.Conv2d(3,64,kernel_size=3,padding=1),
               nn.ReLU(inplace=True),
               nn.MaxPool2d(kernel_size=2,stride=2),
               nn.Conv2d(64, 128, kernel_size=3, padding=1),
               nn.ReLU(inplace=True),
               nn.MaxPool2d(kernel_size=2, stride=2),
               nn.Conv2d(128, 256, kernel_size=3, padding=1),
               nn.ReLU(inplace=True),
               nn.Conv2d(256, 256, kernel_size=3, padding=1),
```

```
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Conv2d(256, 512, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(512,512, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2)
            )
        #此处定义我们的CBAM的注意力机制的模块
            self.attention=cbam_block(512)
        #此处定义我们的第二个特征提取的模块主要就是将CBAN的输出，继续进行特征的提取。
            self.features2=nn.Sequential(
                nn.Conv2d(512, 512, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(512, 512, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2))
        #定义我们分类的模块，主要就是将前面的第二个特征的层的 输出作为输入，并通过全连接层不断的将其模型的输入压缩到指定的输出的维度
            self.classifier=nn.Sequential(
                nn.Dropout(p=0.5),
                #随机失活百分之五十，减少过拟合
                nn.Linear(512*7*7,2048),
                nn.ReLU(inplace=True),
                nn.Dropout(p=0.5),
                nn.Linear(2048,2048),
                nn.ReLU(inplace=True),
                nn.Linear(2048,clss_num)
            )
            if init_weights:
                self._initialize_weights()

        def forward(self,x):
            x=self.features(x)
            x=self.attention(x)
            x=self.features2(x)
            x=torch.flatten(x , start_dim=1)
            x=self.classifier(x)
            return x

        # 初始化我们的权重
        def _initialize_weights(self):
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    #nn.init.kaiming_normal_(m.weight, mode='fan_out', )
                    nn.init.xavier_uniform_(m.weight)
                    if m.bias is not None:
                        nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.Linear):
                     nn.init.xavier_uniform_(m.weight)
                     nn.init.constant_(m.bias,0)
```

5.引入注意力机制CBAM：

```
class ChannelAttention(nn.Module):
    def __init__(self, in_planes, ratio=8):
        super(ChannelAttention, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)
        #通道注意力，这里主要是通过定义申明俩个最大池化和平均池化
        #利用1x1卷积代替全连接
        self.fc1   = nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False)
        self.relu1 = nn.ReLU()
        self.fc2   = nn.Conv2d(in_planes // ratio, in_planes, 1, bias=False)

        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = self.fc2(self.relu1(self.fc1(self.avg_pool(x))))
        max_out = self.fc2(self.relu1(self.fc1(self.max_pool(x))))
        out = avg_out + max_out
        return self.sigmoid(out)

class SpatialAttention(nn.Module):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()

        assert kernel_size in (3, 7), 'kernel size must be 3 or 7'
        padding = 3 if kernel_size == 7 else 1
        self.conv1 = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = torch.mean(x, dim=1, keepdim=True)
        max_out, _ = torch.max(x, dim=1, keepdim=True)
        x = torch.cat([avg_out, max_out], dim=1)
        x = self.conv1(x)
        return self.sigmoid(x)

class cbam_block(nn.Module):
    def __init__(self, channel, ratio=8, kernel_size=3):
        super(cbam_block, self).__init__()
        self.channelattention = ChannelAttention(channel, ratio=ratio)
        self.spatialattention = SpatialAttention(kernel_size=kernel_size)

    def forward(self, x):
        x = x * self.channelattention(x)
        x = x * self.spatialattention(x)
        return x
```

模型的训练：

```
net=MYnet(clss_num=2)

    net.to(device)
    loss_function = nn.CrossEntropyLoss()
    optimizer = optim.Adam(net.parameters(), lr=0.0001)

    epochs = 30
    save_path = './MyNet.pth'
    best_acc = 0.0
    train_steps = len(train_loader)
    val_steps = len(validate_loader)
    Loss_list = []
    Val_acc = []
    Val_loss = []
    Train_acc=[]
    for epoch in range(epochs):
        # train
        net.train()
        running_loss = 0.0
        train_bar = tqdm(train_loader, file=sys.stdout)
        train_acc = 0.0
        for step, data in enumerate(train_bar):
            images, labels = data
            optimizer.zero_grad()
            outputs = net(images.to(device))
            predict_y = torch.max(outputs, dim=1)[1]
            train_acc += torch.eq(predict_y,labels.to(device)).sum().item()
            loss = loss_function(outputs, labels.to(device))
            loss.backward()
            optimizer.step()


            running_loss += loss.item()

            train_bar.desc = "train epoch[{}/{}] loss:{:.3f}".format(epoch + 1,
                                                                     epochs,
                                                                     loss)

        # 验证
        running_val_loss=0.0
        net.eval()
        acc = 0.0
        with torch.no_grad():
            val_bar = tqdm(validate_loader, file=sys.stdout)
            for val_data in val_bar:
                val_images, val_labels = val_data
                outputs = net(val_images.to(device))
                val_loss = loss_function(outputs,  val_labels.to(device))
                predict_y = torch.max(outputs, dim=1)[1]
                acc += torch.eq(predict_y, val_labels.to(device)).sum().item()
```

```
                running_val_loss += val_loss.item()
```

模型的评估和训练的过程可视化：

```
print('Finished Training')
    print("开始绘制训练曲线：----------------------------")
    x1 = np.arange(1, len(Loss_list) + 1)
    y1 = Val_loss
    plt.plot(x1, y1, 'o-', color='blue', label='Val_loss')
    plt.plot(x1, Loss_list, '--', color='red', label='Train_loss')
    plt.title('comparison of model losses')
    plt.ylabel('Loss')
    plt.xlabel('epoch')
    plt.legend()
    plt.show()

    x1 = np.arange(1, len(Val_acc) + 1)
    y1 =Train_acc
    plt.plot(x1, y1, 'o-', color='blue', label='Train_acc')
    plt.plot(x1, Val_acc, '--', color='red', label='Val_acc')
    plt.title('comparison of model acc')
    plt.ylabel('acc')
    plt.xlabel('epoch')
    plt.legend()
    plt.show()
```

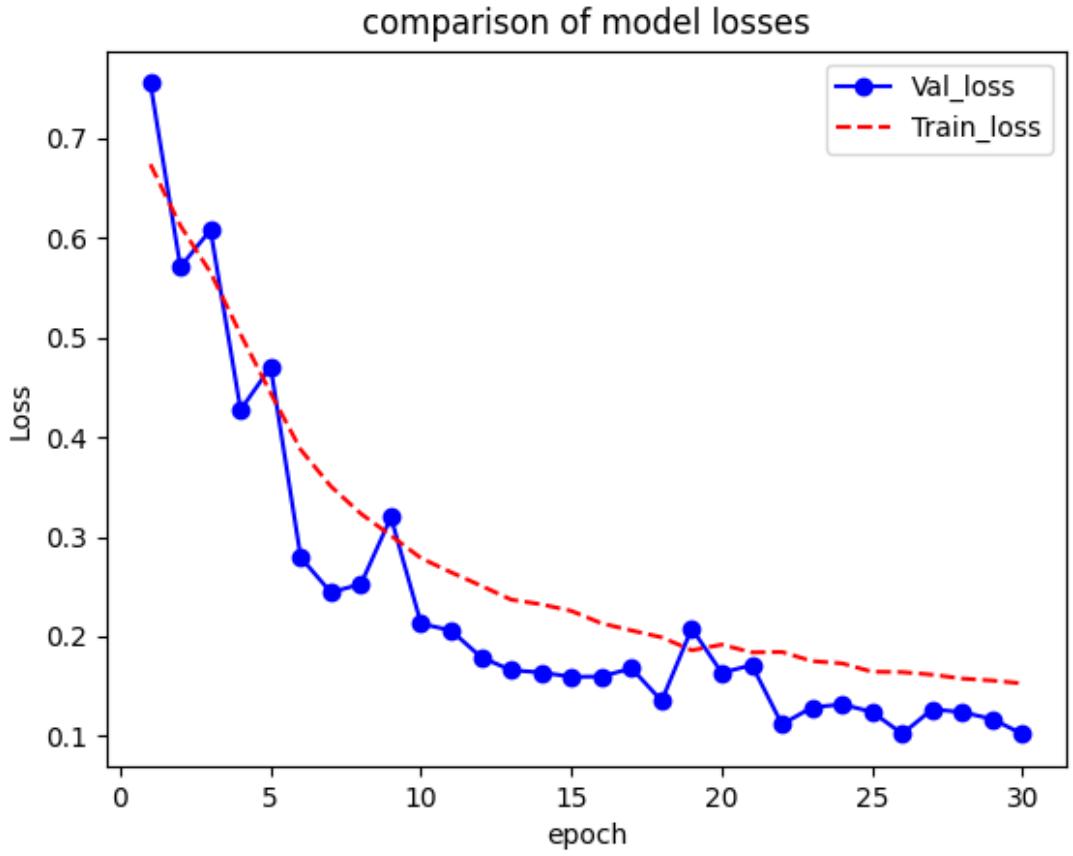
训练结果：

```
100%|██████████| 549/549 [00:09<00:00, 55.91it/s]
[epoch 23] train_loss: 0.175  val_loss: 0.129  val_accuracy: 0.943 train_accuracy: 0.923
train epoch[24/30] loss:0.004: 100%|██████████| 1426/1426 [03:12<00:00,  7.40it/s]
100%|██████████| 549/549 [00:09<00:00, 56.28it/s]
[epoch 24] train_loss: 0.173  val_loss: 0.132  val_accuracy: 0.952 train_accuracy: 0.924
train epoch[25/30] loss:0.553: 100%|██████████| 1426/1426 [03:12<00:00,  7.40it/s]
100%|██████████| 549/549 [00:09<00:00, 55.77it/s]
[epoch 25] train_loss: 0.165  val_loss: 0.124  val_accuracy: 0.948 train_accuracy: 0.929
train epoch[26/30] loss:0.000: 100%|██████████| 1426/1426 [03:12<00:00,  7.40it/s]
100%|██████████| 549/549 [00:09<00:00, 56.20it/s]
[epoch 26] train_loss: 0.164  val_loss: 0.102  val_accuracy: 0.959 train_accuracy: 0.928
train epoch[27/30] loss:0.000: 100%|██████████| 1426/1426 [03:12<00:00,  7.40it/s]
100%|██████████| 549/549 [00:09<00:00, 56.12it/s]
[epoch 27] train_loss: 0.162  val_loss: 0.127  val_accuracy: 0.948 train_accuracy: 0.928
train epoch[28/30] loss:0.081: 100%|██████████| 1426/1426 [03:12<00:00,  7.41it/s]
100%|██████████| 549/549 [00:09<00:00, 55.88it/s]
[epoch 28] train_loss: 0.158  val_loss: 0.124  val_accuracy: 0.946 train_accuracy: 0.931
train epoch[29/30] loss:0.027: 100%|██████████| 1426/1426 [03:12<00:00,  7.41it/s]
100%|██████████| 549/549 [00:09<00:00, 56.16it/s]
[epoch 29] train_loss: 0.156  val_loss: 0.117  val_accuracy: 0.951 train_accuracy: 0.932
train epoch[30/30] loss:0.000: 100%|██████████| 1426/1426 [03:12<00:00,  7.41it/s]
100%|██████████| 549/549 [00:09<00:00, 56.26it/s]
[epoch 30] train_loss: 0.153  val_loss: 0.102  val_accuracy: 0.959 train_accuracy: 0.934
Finished Training
```

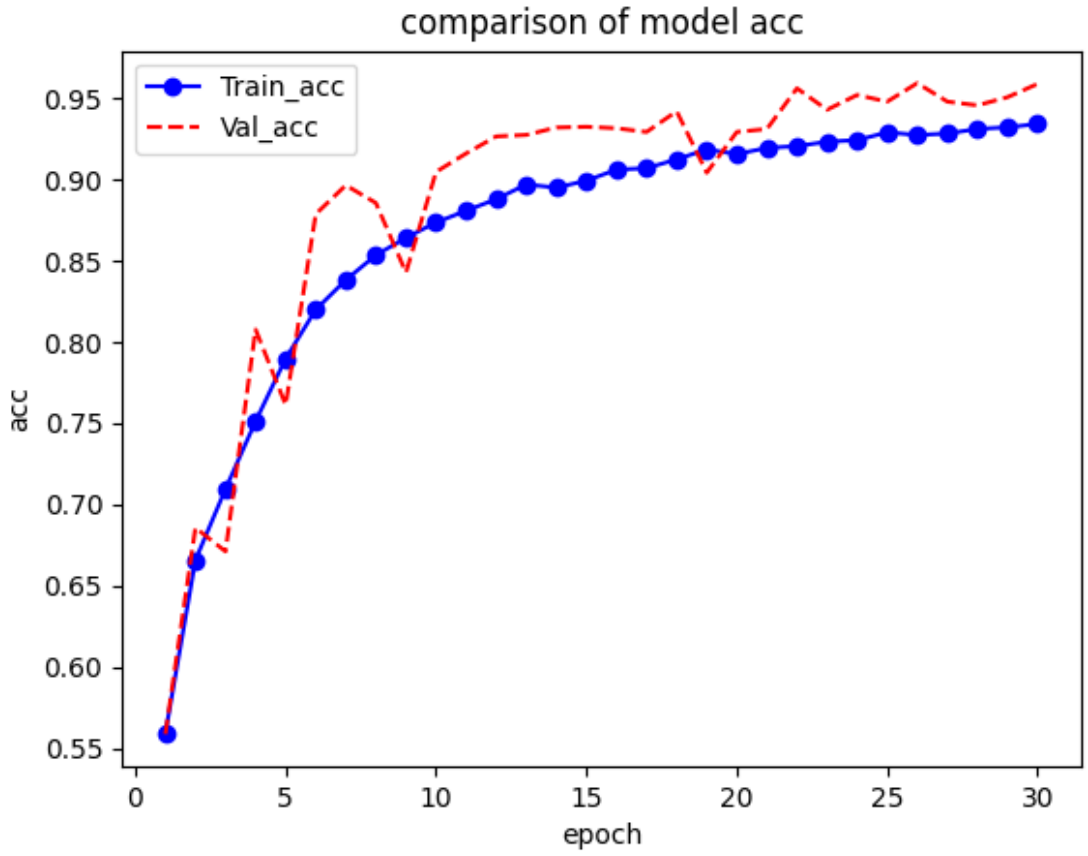
通过三十次的迭代，网络的训练误差和验证误差以及训练准确率和验证准确率如下表：

	loss	**acc**
train	0.153	0.934
Val	0.102	0.959

训练损失可视化图：


comparison of model losses
Val_loss
Train_loss
Loss
epoch
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30

训练准确率变化可视化：


comparison of model acc
Train_acc
Val_acc
acc
epoch
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
5
10
15
20
25
30

实验结论：

通过实验过程，学习到了网络的搭建的方法以及流程，此外我在原先的网络上加入CBAM的注意力机制模块，使得网络的性能大幅度的提升，最终的分类的精度达到了预期，其训练集的准确率为0.934，验证集的准确率为0.959.